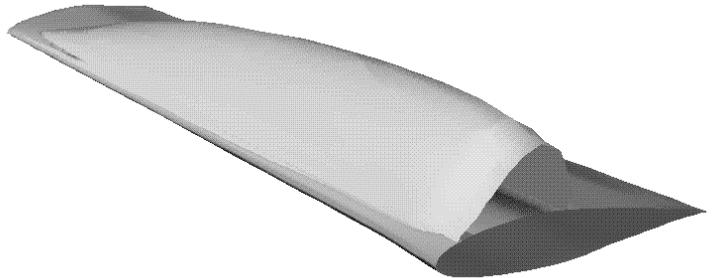


CUMULVS

Hands-On Notes

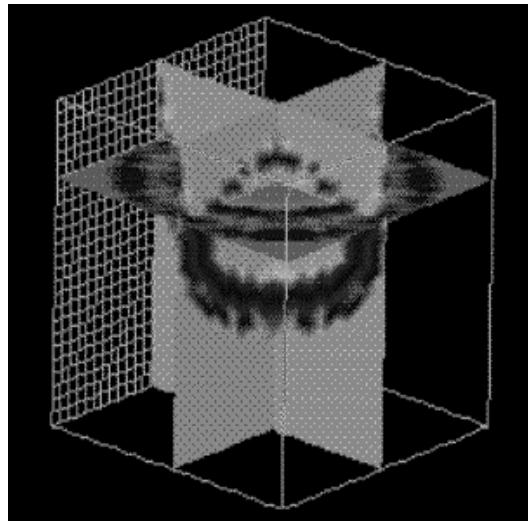


ACTS Toolkit Workshop

Dr. James Arthur Kohl

Computer Science and Mathematics Division

Oak Ridge National Laboratory



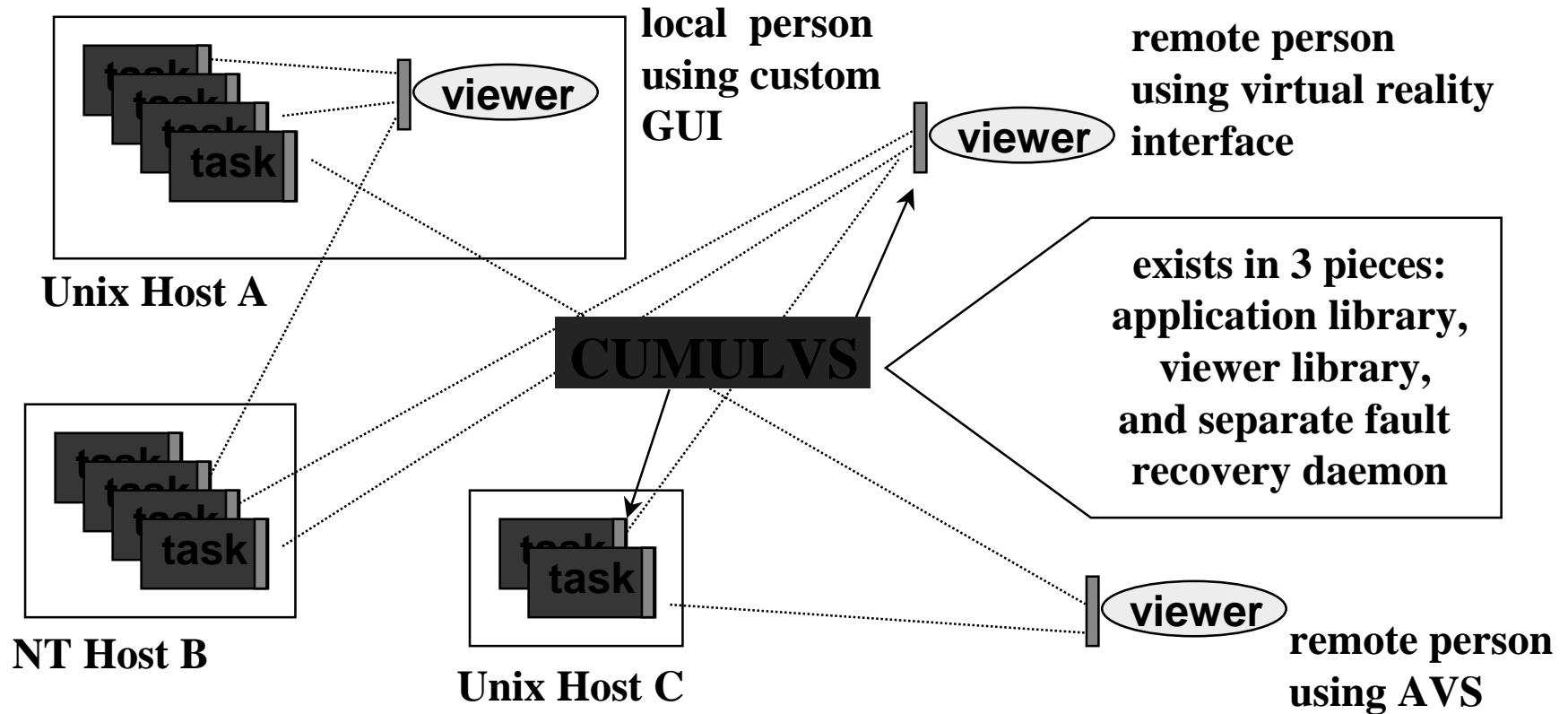
October 11, 2001

Research sponsored by the Applied Mathematical Sciences Research Program, Office of Mathematical, Information, and Computer Sciences, U.S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

Kohl/2000-1

CUMULVS Architecture

coordinate the consistent collection and dissemination of information to / from parallel tasks to multiple viewers



interact with distributed / parallel application or simulation
supports most target platforms (PVM / MPI, Unix / NT, etc.)

Downloading Latest CUMULVS Software Distribution

- Go To CUMULVS Web Site:

<http://www.csm.ornl.gov/cs/cumulvs.html>

or

<http://www.netlib.org/cumulvs>

→ *.tar.gz, *.tar.Z, *.tar.z.uu, *.zip

CUMULVS Environment

- Set STV_ROOT Environment Variable:

or csh: setenv STV_ROOT /home/user/CUMULVS
Bash: export STV_ROOT=/home/user/CUMULVS

- Use Sample “Makefile.aimk” (examples/src/):

```
STV_VIS = NONE
STV_MP = PVM
include $(STV_ROOT)/src/Makeincl.stv
→ $(STV_CFLAGS), $(STV_LDFLAGS), $(STV_LIBS)...
```

- Include CUMULVS Header Files (include/):

⇒ Defines Constants, Functions, Types

C: #include <stv.h>

Fortran: include 'fstv.h'

Instrumenting Programs for CUMULVS

- CUMULVS Initialization ~ One Call (Each Task)
 - ⇒ Logical Application Name, # of Tasks
- Data Fields (Visualization & Checkpointing)
 - ⇒ Local Allocation: Name, Type, Size, Offsets
 - ⇒ Data Distribution: Dim, Decomp, PE Topology
- Steering Parameters
 - ⇒ Logical Name, Data Type, Data Pointer
- Periodic CUMULVS Handler
 - ⇒ Pass Control for Transparent Access / Processing
- Typically 10s of Lines of Code...

CUMULVS Initialization

- C: `stv_init(char *appname, int msgtag,
int ntasks, int nodeid);`
- Fortran: `stvinit(appname, msgtag, ntasks, nodeid)`

where:

- appname ~ logical name of application
- msgtag ~ message tag to reserve for CUMULVS*
- ntasks ~ number of tasks in application
- nodeid ~ task index of the caller

E.g.: `stv_init("solver", 123, 32, 3);`

* For Backwards Compatibility with PVM 3.3 (Before Message Contexts...).

Distributed Data Decomposition

- C: `int decompId = stv_decompDefine(int dataDim,
int *axisType, int *axisInfo1, int *axisInfo2,
int *glb, int *gub, int prank, int *pshape);`
- Fortran: `stvfdecompdefine(dataDim, axisType, axisInfo1,
axisInfo2, glb, gub, prank, pshape, decompId)`

where:

- **decompId** ~ integer decomp handle returned by CUMULVS
- **dataDim** ~ dimensionality of data decomposition
- **axisType** ~ decomp type identifier, for each axis
 - * **stvBlock**, **stvCyclic**, **stvExplicit**, **stvCollapse**
- **axisInfo1,2** ~ specific decomposition details, per axis
 - * E.g. axisInfo1 = Block Size or Cycle Size
 - * E.g. axisInfo1 = Explicit Lower Bound, axisInfo2 = Upper Bound
 - * Note: axisInfo1,2 can be set to **stvDefaultInfo**

Data Decomposition (cont.)

Global Bounds and Pshape

- C: `stv_decompDefine(...,`
`int *glb, int *gub, int prank, int *pshape);`
- Fortran: `stvfdecompdefine(..., glb, gub,`
`prank, pshape)`

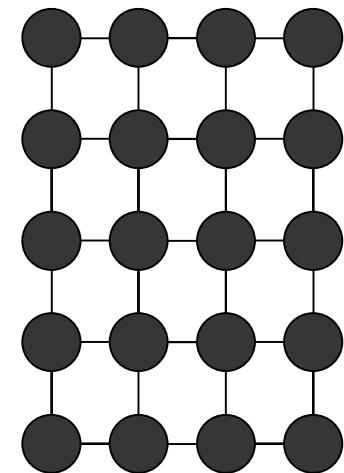
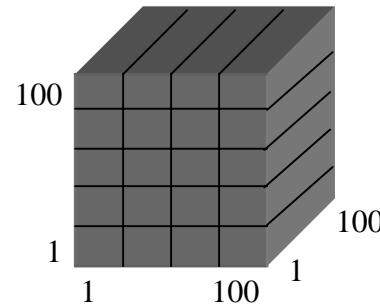
where:

- **glb** ~ lower bounds of decomp in global coordinates
- **gub** ~ upper bounds of decomp in global coordinates
- **prank** ~ dimensionality of processor topology (pshape)
- **pshape** ~ logical processor organization / topology
 - * number of processors per axis

Example Decomposition #1

3D Standard Block Decomp on 2D Processor Array

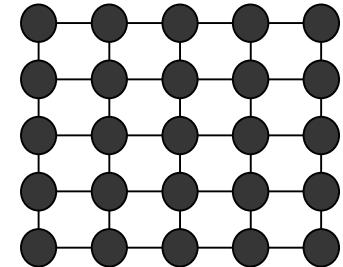
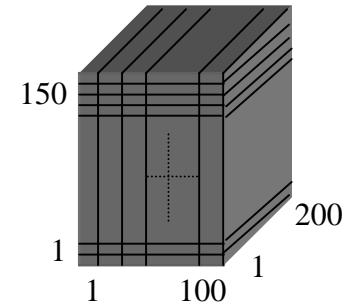
```
int decompId, axisType[3], glb[3], gub[3], pshape[2];  
  
axisType[0] = axisType[1] = stvBlock;  
axisType[2] = stvCollapse;  
  
glb[0] = glb[1] = glb[2] = 1;  
gub[0] = gub[1] = gub[2] = 100;  
  
pshape[0] = 4; /* "Standard" Block Size = 25 */  
pshape[1] = 5; /* "Standard" Block Size = 20 */  
  
decompId = stv_decompDefine( 3,  
    axisType, stvDefaultInfo, stvDefaultInfo,  
    glb, gub, 2, pshape );u
```



Example Decomposition #2

3D Block-Cyclic Decomp on 2D Processor Array

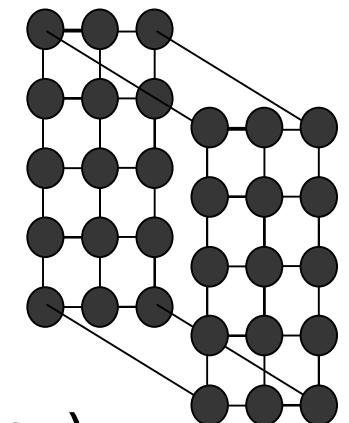
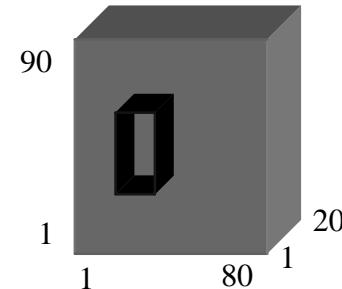
```
int axisType[3], axisInfo[3], glb[3], gub[3], pshape[2];  
  
axisType[0] = stvBlock;    axisInfo[0] = 10; /* Block */  
axisType[1] = stvCyclic;   axisInfo[1] = 3;  /* Cycle */  
axisType[2] = stvCollapse;  
  
glb[0] = glb[1] = glb[2] = 1;  
gub[0] = 100;  gub[1] = 150;  gub[2] = 200;  
  
pshape[0] = 5;  pshape[1] = 5;  
  
decompId = stv_decompDefine( 3, axisType,  
                           axisInfo, stvDefaultInfo, glb, gub, 2, pshape );
```



Example Decomposition #3

3D Explicit Decomp on 3D Processor Array

```
int axisType[3], axisInfo1[3], axisInfo2[3],... pshape[3];  
  
axisType[0] = axisType[1] = axisType[2] = stvExplicit;  
axisInfo1[0] = 11;   axisInfo2[0] = 17;  
axisInfo1[1] = 33;   axisInfo2[1] = 57;  
axisInfo1[2] = 1;    axisInfo2[2] = 7;  
  
glb[0] = glb[1] = glb[2] = 1;  
gub[0] = 80;   gub[1] = 90;   gub[2] = 20;  
  
pshape[0] = 3;   pshape[1] = 5;   pshape[2] = 2;  
  
decompId = stv_decompDefine( 3, axisType,  
                           axisInfo1, axisInfo2, glb, gub, 3, pshape );
```



Additional Explicit Bounds

For Extra Explicit Array Subregions Per Task

- C: `int stv_decompAddExplicitBounds(int decompId,
int axis, int lowerBound, int upperBound);`
- Fortran: `stvfdecompaddexplicitbounds(decompId,
axis, lowerBound, upperBound, info)`

where:

- decompId ~ integer handle to CUMULVS decomp
- axis ~ index of desired decomposition axis
- lowerBound ~ lower bound of subregion (global coords)
- upperBound ~ upper bound of subregion (global coords)

Additional Explicit Bounds – Vector Version

For Multiple Additional Explicit Array Subregions

- C: `int stv_decompAddExplicitVBounds(int decompId,
int axis, int *lowerBounds, int *upperBounds,
int nbounds);`
- Fortran: `stvfdecompaddexplicitvbounds(decompId,
axis, lowerBounds, upperBounds, nbounds, info)`

where:

- lowerBounds ~ array of lower bounds (global coords)
- upperBounds ~ array of upper bounds (global coords)

Example Decomposition #4

3D Local Array

```
int decompId, glb[3], gub[3];  
  
glb[0] = glb[1] = glb[2] = 0;  
gub[0] = gub[1] = gub[2] = 100;  
  
decompId = stv_decompDefine( 3,  
                             stvLocalArray, stvLocalArray, stvLocalArray,  
                             glb, gub, 1, stvLocalArray );
```

Data Field Definition

- C: `int fieldId = stv_fieldDefine(STV_VALUE var,
char *name, int decompId, int *arrayOffset,
int *arrayDecl, int type, int *paddr, int aflag);`
- Fortran: `stvffielddefine(var, name, decompId,
arrayOffset, arrayDecl, type, paddr, aflag, fieldId)`

where:

- **fieldId** ~ integer field handle returned by CUMULVS
- **var** ~ reference (pointer) to local data array storage
- **name** ~ logical name of data field
- **decompId** ~ handle to pre-defined decomposition template
- **type** ~ integer constant that defines data type
 - * `stvByte, stvInt, stvFloat, stvDouble, stvLong, stvCplx, stvUint, ...`

Data Field Definition (cont.)

Local Data Field Allocation

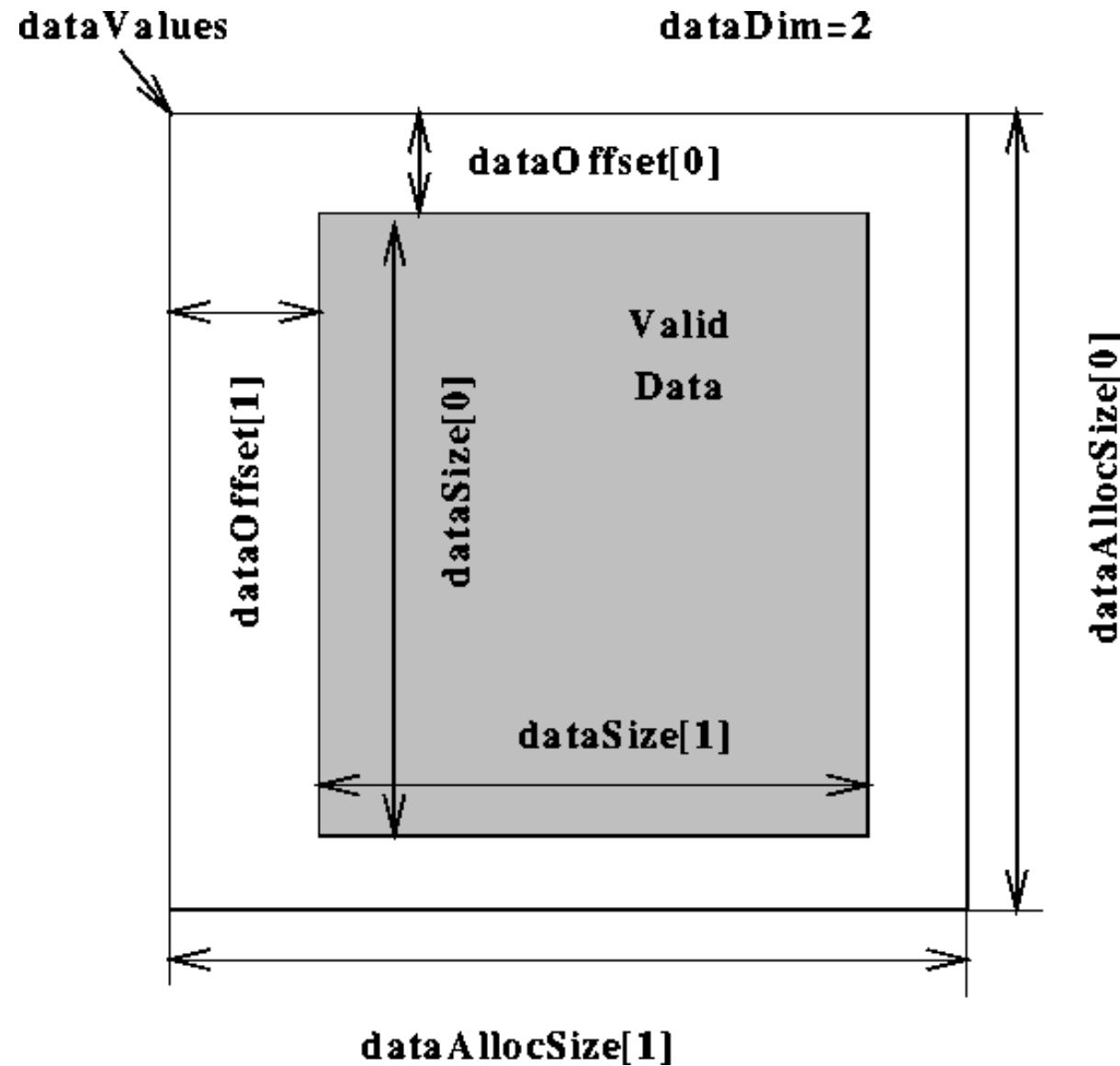
- C: `int fieldId = stv_fieldDefine(...,
 int *arrayOffset, int *arrayDecl, ...);`
- Fortran: `stvffielddefine(..., arrayOffset, arrayDecl ...)`

where:

- **arrayOffset** ~ offset to “valid” data, along each axis
- **arrayDecl** ~ overall allocated array size, along each axis

Note: Size of actual “valid” data is determined in conjunction
with data decomposition information...

Local Allocation Organization



Data Field Definition (cont.)

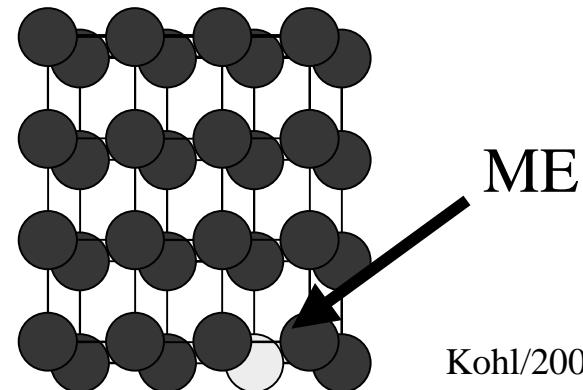
Processor “Address”

- C: `int fieldId = stv_fieldDefine(..., int *paddr, ...);`
- Fortran: `stvffielddefine(..., paddr, ...)`

where:

→ **paddr** ~ integer array containing the local task’s position
in the overall processor topology (pshape)
* paddr index starts from 0!
(NOT the “physical” processor address!)

E.g. for Pshape = { 4, 4, 2 } and Prank = 3, Paddr = { 2, 3, 1 } is:
the 3rd processor of 4 along axis 1
the 4th processor of 4 along axis 2
the 2nd processor of 2 along axis 3



Data Field Definition (cont.)

Access / Usage Flag

- C: `int fieldId = stv_fieldDefine(..., int aflag);`
- Fortran: `stvffielddefine(..., aflag, ...)`

where:

- **aflag** ~ integer flag to indicate external access to field:
- * **stvVisOnly** ~ access data field for visualization only
 - * **stvCpOnly** ~ access data field for checkpointing only
 - * **stvVisCp** ~ access data field for visualization *and* checkpointing

Example Data Field #1

100x100x100 Float Array on Processor { 1, 2 } of Decomp
(E.g., say, as part of 1000x1000x1000 Global Decomp)

```
float foo[100][100][100];
int fieldId, declare[3], paddr[2];

declare[0] = declare[1] = declare[2] = 100;

paddr[0] = 1;   paddr[1] = 2;

fieldId = stv_fieldDefine( foo, "Foo", decompId,
                           stvNoFieldOffsets, declare, stvFloat, paddr,
                           stvVisOnly );
```

Example Data Field #2

80x80x60 Double Array on Processor { 2, 1, 3 } of Decomp
(With 10 Element Neighbor Boundary in Local Allocation)

```
double bar[100][100][80];
int fieldId, offsets[3], declare[3], paddr[3];

offsets[0] = offsets[1] = offsets[2] = 10;

declare[0] = declare[1] = 100; declare[2] = 80;

paddr[0] = 2; paddr[1] = 1; paddr[2] = 3;

fieldId = stv_fieldDefine( bar, "Bar", decompId,
                           offsets, declare, stvDouble, paddr, stvVisCp );
```

Particle Definition

A Container for Particle Data Fields

- C:

```
int particleId = stv_particleDefine( char *name,
                                         int dataDim, int *glb, int *gub, int ntasks,
                                         STV_GET_PARTICLE get_particle,
                                         STV_MAKE_PARTICLE make_particle );
```
- Fortran:

```
stvfparticledefine( name, dataDim, glb, gub,
                           ntasks, get_particle, make_particle, particleId )
```

where:

- **particleId** ~ integer particle handle returned by CUMULVS
- **name** ~ logical name of particle wrapper
- **dataDim** ~ dimensionality of particle space
- **glb, gub** ~ global bounds of particle space
- **ntasks** ~ number of parallel tasks cooperating on particles

Particle Definition (cont.)

User-Defined Particle Functions

- C: `int particleId = stv_particleDefine(...,
STV_GET_PARTICLE get_particle,
STV_MAKE_PARTICLE make_particle);`
- Fortran: `stvfparticledefine(..., get_particle,
make_particle, ...)`

where:

- **get_particle()** ~ user-defined accessor function
 - * returns a user-defined handle to CUMULVS for the given particle
- **make_particle()** ~ user-defined constructor function
 - * allow CUMULVS to create a new particle (checkpoint recovery)
(Can set make_particle() to NULL if not checkpointing)

Get a Particle for CUMULVS

Particle Accessor Function

- C: `void get_particle(int index, STV_REGION region,
STV_PARTICLE_ID *id, int *coords);`
- Fortran: `get_particle(index, region, id, coords)`

where:

- **index** ~ requested particle index
 - * positive integer value, index set to **-1** for search reset
 - * if particle not found for given index, should return **id** as NULL
- **region** ~ subregion of particle space to search
 - * can use `stv_particle_in_region()` utility function...
- **id** ~ user-defined particle handle returned by search
 - * `(void *)` → anything the application wants to cast it to...
- **coords** ~ array of coordinates for returned particle

Make a Particle for CUMULVS

Particle Constructor Function – Checkpoint Recovery

- C: `void make_particle(int *coords,
 STV_PARTICLE_ID *id);`
- Fortran: `make_particle(coords, id)`

where:

- **coords** ~ array of coordinates for requested particle creation
- **id** ~ user-defined particle handle returned for new particle
 - * (void *) → anything the application wants to cast it to...

Particle Field Definition

Actual Data Fields Associated with a Particle

- C: `int pfieldId = stv_pfieldDefine(char *name,
int particleId, int type, int nelems,
STV_GET_PFIELD get_pfield, STV_VALUE get_pfield_arg,
STV_SET_PFIELD set_pfield, STV_VALUE set_pfield_arg,
int aflag);`
- Fortran: `stvfpfielddefine(name, particleId, type, nelems,
get_pfield, get_pfield_arg, set_pfield, set_pfield_arg,
aflag, pfieldId)`

where:

- **pfieldId** ~ returned integer particle field handle
- **name** ~ logical name of particle data field
- **particleId** ~ handle to encapsulating particle wrapper

Particle Field Definition (cont.)

Actual Data Fields Associated with a Particle

- C: `stv_pfieldDefine(..., int type, int nelems, ...,
int aflag);`
- Fortran: `stvfpfielddefine(..., type, nelems, ...,
aflag, ...)`

where:

- **type** ~ data type for particle field (**stvInt**, **stvFloat**, etc.)
- **nelems** ~ number of data elements
 - * only simple 1-D arrays (vectors) are supported as particle fields
- **aflag** ~ same as for `stv_fieldDefine()`, access flag
 - * **stvVisOnly**, **stvCpOnly**, or **stvVisCp**.

Particle Field Definition (cont.)

User-Defined Particle Field Functions

- C: `stv_pfieldDefine(...,`
`STV_GET_PFIELD get_pfield, STV_VALUE get_pfield_arg,`
`STV_SET_PFIELD set_pfield, STV_VALUE set_pfield_arg ...);`
- Fortran: `stvfpfielddefine(...,`
`get_pfield, get_pfield_arg,`
`set_pfield, set_pfield_arg, ...)`

where:

- **get_pfield() / get_pfield_arg** ~ user-defined accessor/arg
 - * return reference to actual data storage
 - * extra user-defined argument, (void *) for vectored get_pfield() impl
- **set_pfield() / set_pfield_arg** ~ user-defined constructor/arg
 - * for checkpoint recovery, set the values of given particle field array
 - * extra user-defined argument, (void *) for vectored set_pfield() impl

Get a Particle Field for CUMULVS

Particle Field Accessor Function

- C: `void get_pfield(STV_PARTICLE_ID id,
STV_VALUE user_data, STV_VALUE *data);`
- Fortran: `get_pfield(id, user_data, data);`

where:

- **id** ~ user-defined particle handle
 - * as returned by previous `get_particle()` call
- **user_data** ~ arbitrary user-defined argument
 - * (`void *`) value for vectored implementation of `get_pfield()`
- **data** ~ reference to actual data storage

E.g. `if (user_data == "density")
*data = ((mystruct) id) ->density;`

Set a Particle Field for CUMULVS

Particle Field Constructor Function – Checkpoint Recovery

- C: `void set_pfield(STV_PARTICLE_ID id,
STV_VALUE user_data, STV_VALUE data);`
- Fortran: `set_pfield(id, user_data, data);`

where:

- **id** ~ user-defined particle handle
 - * as returned by previous `get_particle()` call
- **user_data** ~ arbitrary user-defined argument
 - * `(void *)` value for vectored implementation of `set_pfield()`
- **data** ~ reference to new data values to set

E.g. `if (user_data == "density")
((mystruct) id)->density = data;`

Steering Parameter Definition

- C: `int paramId = stv_paramDefine(char *name,
STV_VALUE var, int type, int aflag);`
- Fortran: `stvfparamdefine(name, var, type, aflag,
paramId)`

where:

- **paramId** ~ returned integer steering parameter handle
- **name** ~ logical name of steering parameter
- **var** ~ reference to actual steering parameter storage
- **type** ~ data type of steering parameter (**stvInt**, **stvFloat**, ...)
- **aflag** ~ external access flag, same as before
 - * **stvVisOnly**, **stvCpOnly** or **stvVisCp**

Steering Parameter Definition

Vector Parameters

- C: `int paramId = stv_vparamDefine(char *name,
STV_VALUE *vars, char **pnames, int *types, int num,
int aflag);`
- Fortran: (no Fortran equivalent yet)

where:

- **paramId** ~ returned integer steering vector handle
- **name** ~ logical name of steering vector
- **vars** ~ array of references to steering element storage
- **pnames** ~ array of logical names for steering elements
- **types** ~ array of steering element data types
 - * OR with **stvIndex** for “indexed” steering vector (**stvInt** | **stvIndex**)
- **aflag** ~ external access flag (**stvVisOnly**, **stvCpOnly**, **stvVisCp**)

Steering Parameter Updates...

- C: `int changed = stv_isParamChanged(int paramId);`
- Fortran: `stvfisparamchanged(paramId, changed)`

where:

- **paramId** ~ integer steering parameter handle
- **changed** ~ status of steering parameter
 - * > 0 indicates an update has occurred
 - * = 0 indicates no change to parameter value
 - * < 0 indicates an error condition (bad parameter id, etc)

Passing Control to CUMULVS

- After Data Fields and Parameters Are Defined
 - ⇒ Single Periodic Call to Pass Control to CUMULVS
 - ⇒ Once Per Iteration? Between Compute Phases?

C: `int params_changed = stv_sendReadyData(`
 `int update_field_times);`

Fortran: `stvfsendreadydata(update_field_times,`
 `params_changed)`

where:

→ **params_changed** ~ return code that indicates whether any steering parameter values have been updated

* > 0 an update has occurred, = 0 no changes to any values

→ **update_field_times** ~ flag to control data field times

* in general, just use **stvSendDefault...** ☺

CUMULVS Checkpoint Initialization

- C: `int cp_restart = stv_cpInit(char *aout_name,
int notify_tag, int *ntasks);`
- Fortran: `stvfcpinit(aout_name, notify_tag, ntasks,
cp_restart)`

where:

- **cp_restart** ~ status value, indicates restart / failure recovery
 - * use to circumvent normal application startup and data initialization!!
- **aout_name** ~ name of executable file to spawn for restart
- **notify_tag** ~ message code for notify messages (PVM only)
- **ntasks** ~ number of tasks in application
 - * in a restart condition, this value can be set by CUMULVS!

Note: `stv_cpInit()` must be called **after** `stv_init()`!

Alternate Checkpoint Restart Check

- C: `int cp_restart = stv_isCpRestart();`
- Fortran: `stvfiscprestart(cp_restart)`

where:

→ **cp_restart** ~ status value, indicates restart / failure recovery
* use to circumvent normal application startup and data initialization!!

(Additional routine to check for Restart / Failure Recovery.)

Rollback Versus Restart...

- Rollback Recovery:
 - ⇒ Only Replace Failed Tasks, “Roll Back” the Rest
 - ⇒ Elegant & Cool, But You Must...
 - Monitor ALL Communication for Restart Notification
 - Unroll Program Stack, Reset Comm & Files...
 - ⇒ Necessary for High Overhead Restart Cases
- Restart Recovery:
 - ⇒ “Genocide” ~ Kill Everything & Restart All Tasks
 - ⇒ Simple Approach, No Additional Instrumentation
 - ⇒ Not As Efficient A Recovery In All Cases...

CUMULVS Option Interface...

Checkpointing Option

- One Option As Part of Larger Interface...
- Invoke **Before** You Call `stv_cpInit()`! ☺

```
/* Set Recovery Option to Rollback */
stv_setopt( stvDefault, stvOptCpRecovery,
            stvOptCpRollback );
or
/* Set Recovery Option to Restart */
stv_setopt( stvDefault, stvOptCpRecovery,
            stvOptCpRestart );
```

Same in Fortran:

```
stvfsetopt( STVDEFAULT, STVOPTCPRECOVERY,
            STVOPTCPRESTART ) . . .
```

Checkpoint Collection

- Actually Collect Data from Local Task
- Invoke When Parallel Data / State Consistent
 - ⇒ Highly Non-Trivial in General!! (Chandy/Lamport)
 - ⇒ Straightforward for Most Iterative Applications
 - Save Checkpoint at Beginning or End of Main Loop

- C: `int status = stv_checkpoint();`
- Fortran: `stvfcheckpoint(status)`

where:

→ **status** < 0 indicates system failure

Control Overhead Using Checkpointing Frequency:

```
if ( !(i % 100) ) stv_checkpoint();
```

Restoring Data From A Checkpoint

- After Restart Condition Has Been Identified...
- Can Be Invoked Incrementally to Bootstrap Data
 - ⇒ Interlace With Data Field / Parameter Definitions...
 - ⇒ Automatically Loads Data Back Into User Memory for All Defined Data Fields and Parameters
 - C: `int remaining = stv_loadFromCP();`
 - Fortran: `stvfloadfromcp(remaining)`

where:

→ **remaining** ~ number of remaining data fields or parameters to define / restore

Example Instrumentation

Restart from a Checkpoint

```
/* Check Restart Status... (Returned By stv_cpInit( )) */
/* Load Program Variables From Checkpoint Data */
if ( restart )  stv_loadFromCP( );

/* Allocate "indices" Vector Using "del" as Size */
ix = (int *) malloc( ( 100.0 / del ) * sizeof(int) );

/* Define "indices" Parameter for CUMULVS */
stv_paramDefine( "indices", ix[], stvInt, stvVisCp );

/* Load Newly-Defined Program Vars (ix[]) from Ckpt */
if ( restart )  stv_loadFromCP( );
else  user_init_data( );
```

Oh Yeah, One Last Routine... ☺

Finished Checkpointing

• • •

```
/* Tell CUMULVS to Stop Checkpoint Recovery. */
/* (So Task Can Exit Normally & R.I.P. :-) */
stv_cpFinished( );

exit( );
}
```

CUMULVS Summary

- Interact with Scientific Simulations
 - ⇒ Dynamically Attach Multiple Visualization Front-Ends
 - ⇒ Steer Model & Algorithm Parameters On-The-Fly
 - ⇒ Automatic Heterogeneous Fault Recovery & Migration
- Future Opportunities
 - ⇒ Couple Disparate Simulation Models
 - ⇒ Integrate as “MxN” Component in CCA
 - ⇒ Application Instrumentation GUI / Pre-Compiler

<http://www.csm.ornl.gov/cs/cumulvs.html>

Email: cumulvs@msr.csm.ornl.gov